Algorithms and Data Structures 2
CS 1501

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Homework 7: this Friday @ 11:59 pm
  - Lab 6: next Monday 10/31 @ 11:59 pm
  - **Assignment 2: Friday 11/4 @ 11:59 pm**
  - Lab 7: Monday 11/7 @ 11:59 pm
- Live Support Session for Assignment 2
  - This Friday 7-8 pm (https://pitt.zoom.us/my/khattab)
- Weekly Live QA Session on Piazza
  - Friday 4:30-5:30 pm

Previous lecture

- ADT Priority Queue (PQ)
  - Heap implementation
- Heap Sort
- Indexable PQ

This Lecture

- ADT Graph
  - definitions
  - representations
  - traversals

Problem of the Day

- **Input**: A file containing LinkedIn (LI) accounts and their connections
  - Account1: Connection1, Connection2, …
  - Account2: Connection1, Connection2, …
  - …

Problem of the Day

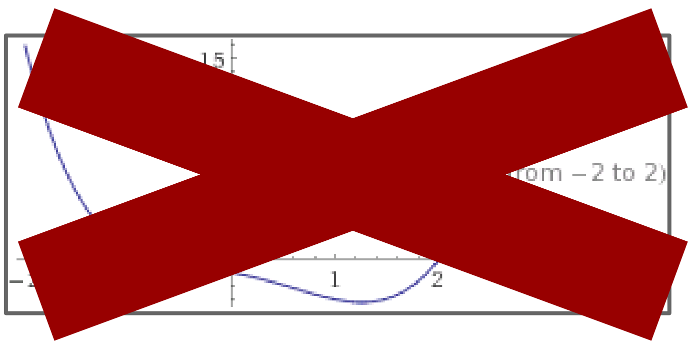
- **Output**: Answer the following questions:
  - Given two LI accounts, how "far" are they from each other?
    - e.g., 1st connection?, 2nd connection?, etc.
  - Are the accounts in the file all ***connected***?
    - If not, how many ***connected components*** are there?
  - For each connected component, are there certain accounts that if removed, the remaining accounts become ***partitioned***?

Which Data Type to use?

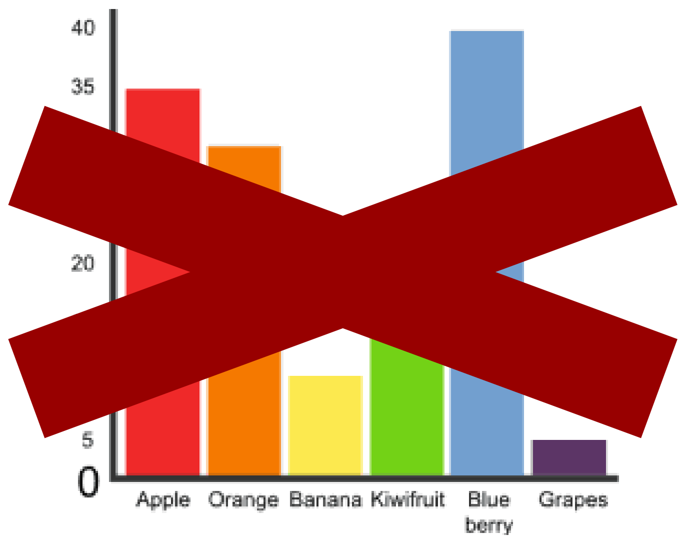
- Let's think first about how to organize the data that we have in memory
- Note that the operations are different from what we have been used to (search, sort, min, max, add, delete, …)

- Account1: Connection1, Connection2, …
- Account2: Connection1, Connection2, …
- …

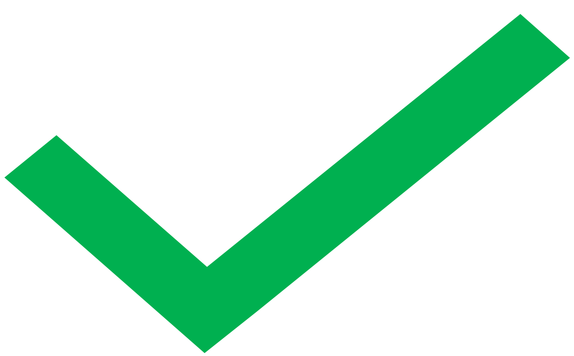
Graphs!

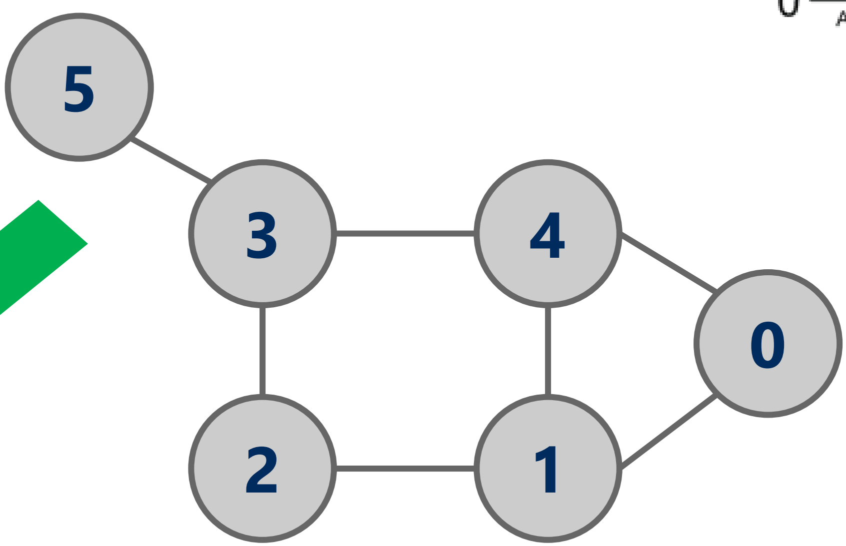


15
1
2



40
35
20
5
0
Apple
Orange
Banana
Kiwifruit
Blue berry
Grapes

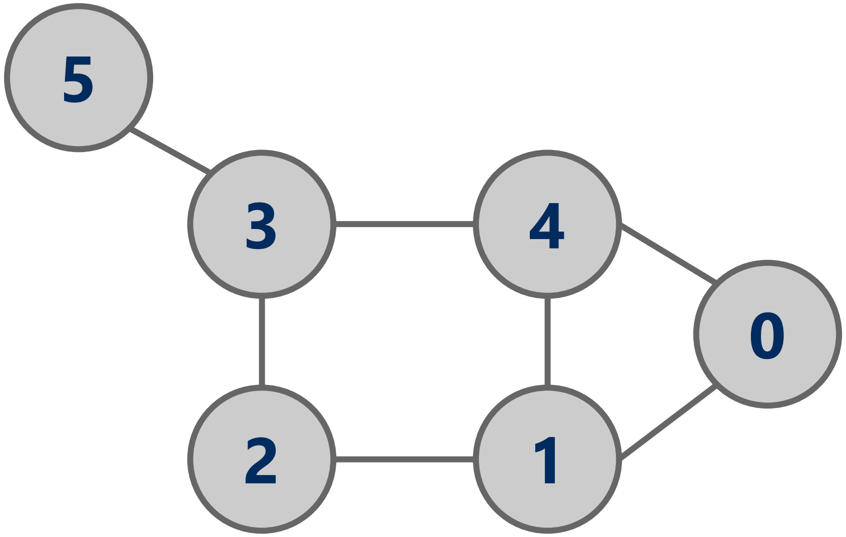




5
3
4
0
2
1

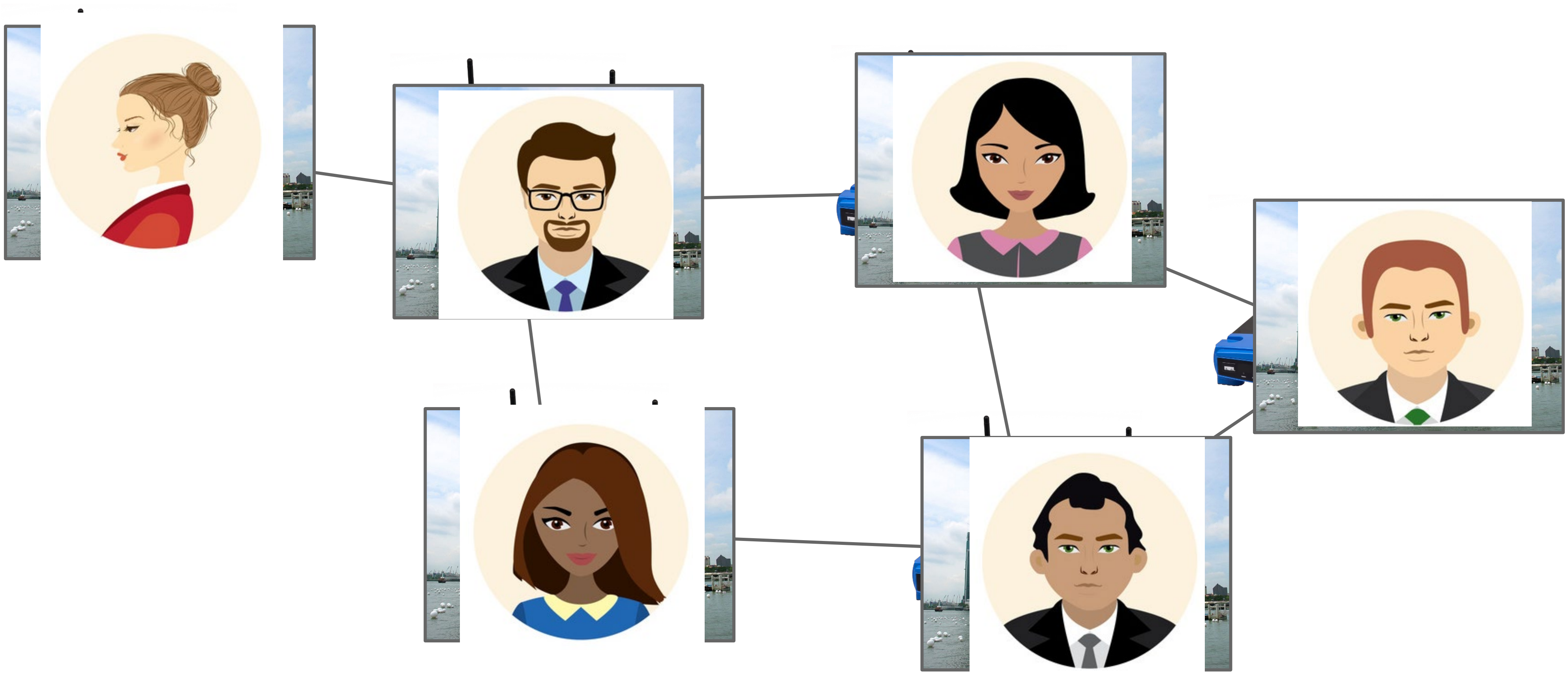
Graphs

- A graph G = (V, E)
  - where V is a set of vertices
  - E is a set of edges connecting vertex pairs
- Example:
  - V = {0, 1, 2, 3, 4, 5}
  - E = {(0, 1), (0, 4), (1, 2), (1, 4), (2, 3), (3, 4), (3, 5)}



Why?

- Can be used to model many different scenarios

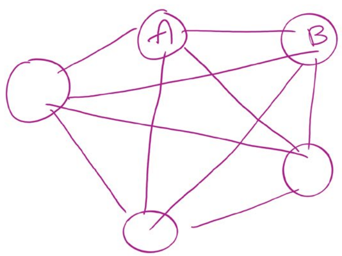


Some definitions

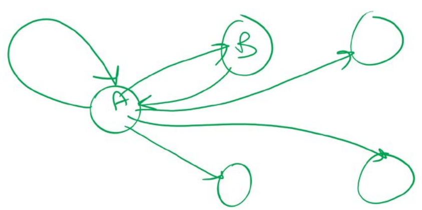
- Undirected graph
  - Edges are unordered pairs: (A, B) == (B, A)
- Directed graph
  - Edges are ordered pairs: (A, B) != (B, A)
- Adjacent vertices, or neighbors
  - Vertices connected by an edge

Graph sizes

- Let v = |V|, and e = |E|
- Given v, what are the minimum/maximum sizes of e?
  - Minimum value of e?
    - Definition doesn't necessitate that there are any edges…
    - So, 0
  - Maximum of e?
    - Depends…
      - Are self edges allowed?
      - Directed graph or undirected graph?
    - In this class, we'll assume directed graphs have self edges while undirected graphs do not



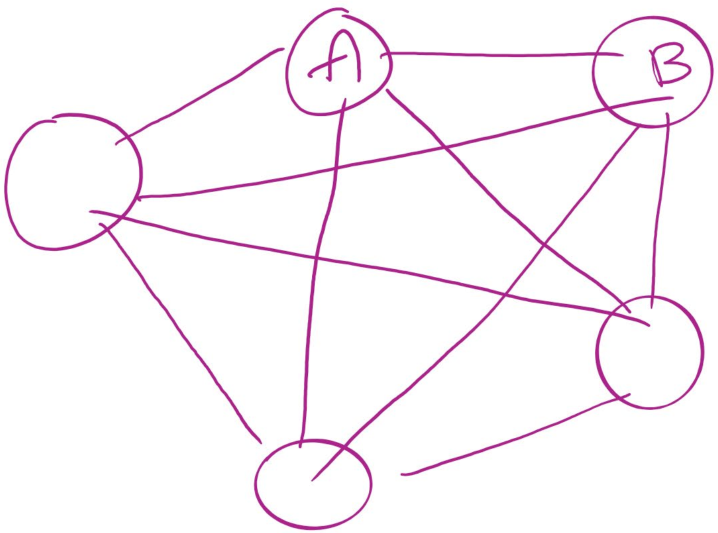
$$\frac{v(v-1)}{2}$$



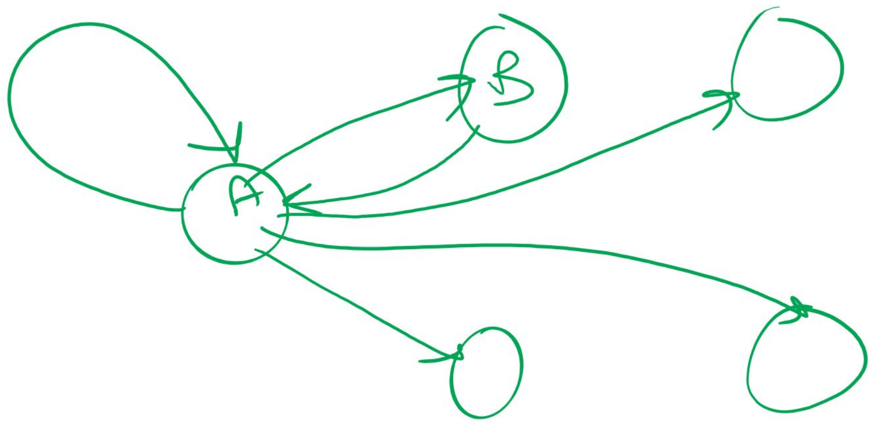
$$v * v = v^2$$

Maximum value of e (MAX)

- Undirected graph
  - no self edges
  - v*(v-1)?
  - But, A->B is the same edge as B-> A
  - Are we counting each twice?
  - v*(v-1)/2

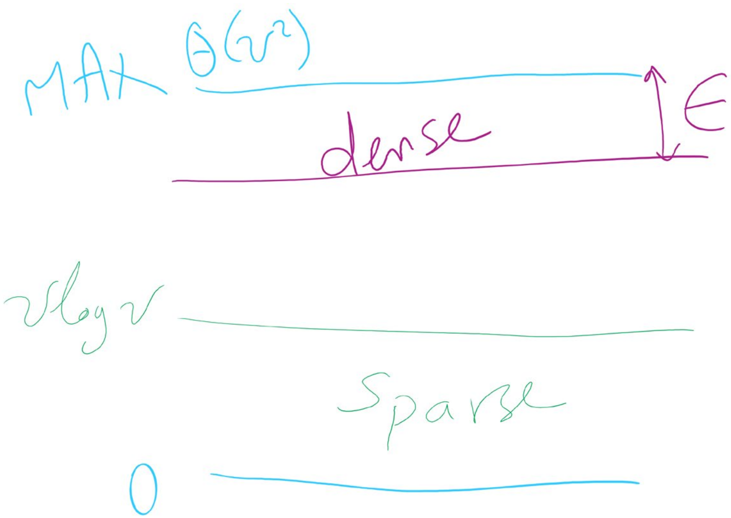


- Directed graph
  - self edges allowed
  - v*v?
  - A -> B is a different edge than B -> A
  - v^2

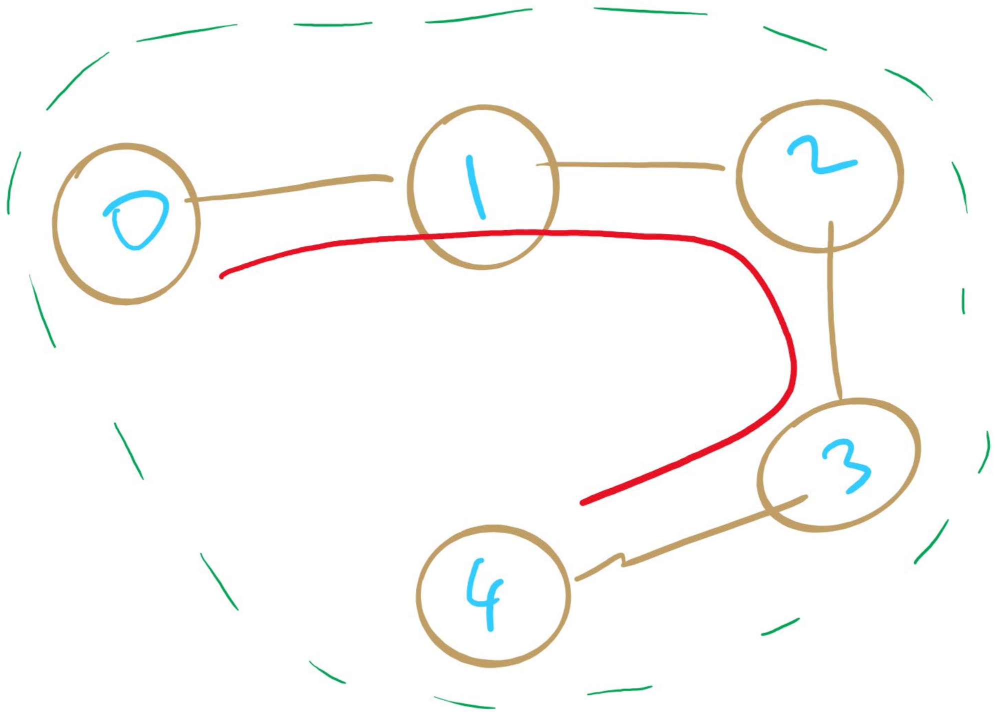


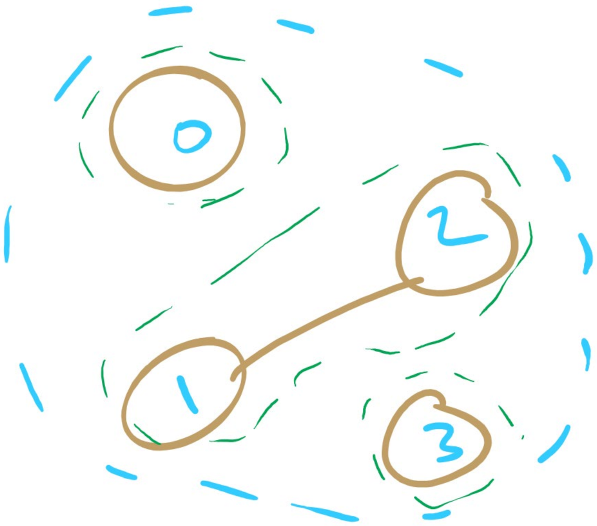
More definitions

- A graph is considered *sparse* if:
  - e <= v lg v
- A graph is considered *dense* as it approaches the maximum number of edges
  - I.e., e == MAX - ε
- A *complete* graph has the maximum number of edges
- Have we seen “sparse” and dense before?



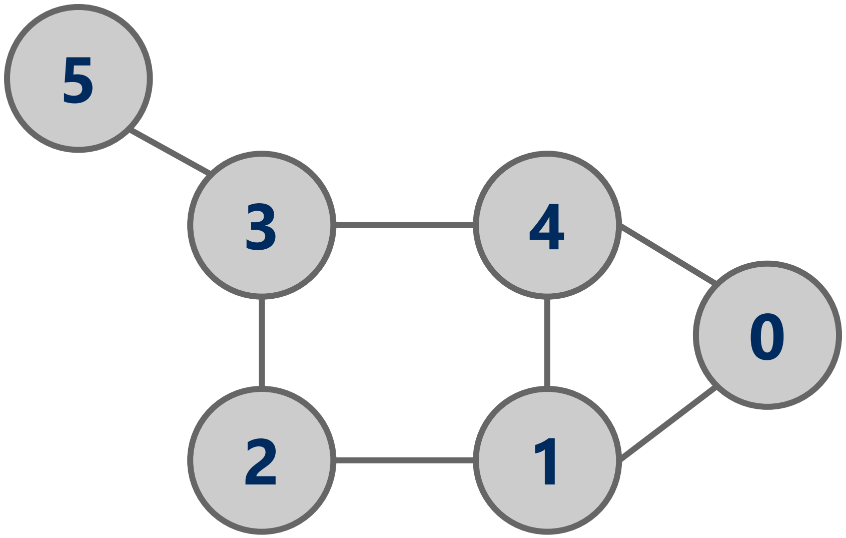
Sparse graphs



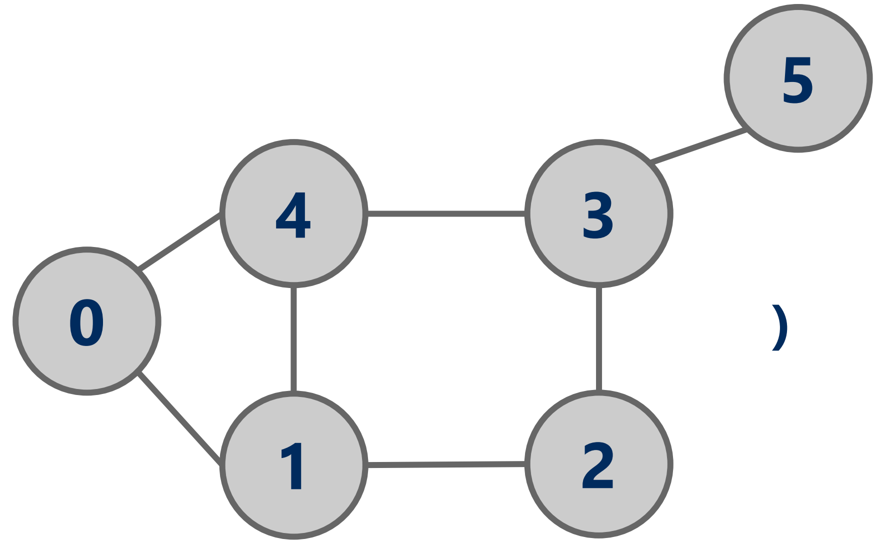


Question:

- Is



.equals(



)

Representing graphs

- Trivially, graphs can be represented as:
  - List of vertices
  - List of edges
- Performance?
  - Assume we're going to be analyzing static graphs
    - I.e., no insert and remove
  - So what operations should we consider?

Graph operations

- Static graphs
  - check if two vertices are neighbors
  - find the list of neighbors of a given vertex
    - for directed graphs, in-neighbors and out-neighbors
- Dynamic graphs
  - add/remove edges
  - Not our focus in this class

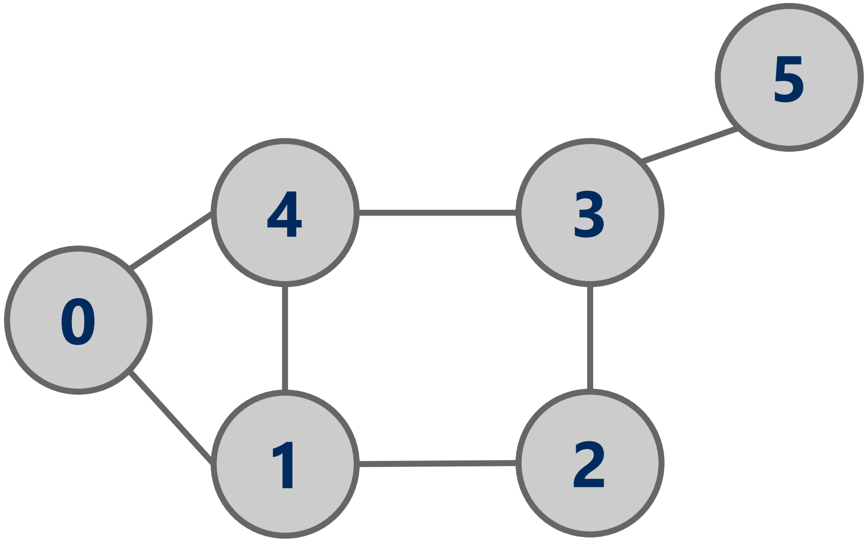
Representing graphs

- Trivially, graphs can be represented as:
  - List of vertices
  - List of edges
- Performance?
  - Check if two vertices are neighbors
    - $O(e)$
  - Find the list of neighbors of a given vertex
    - $O(e)$
- Space?
  - $\Theta(v + e)$ memory

Using an adjacency matrix

- Rows/columns are vertex labels
  - M[i][j] = 1 if $(i, j) \in E$
  - M[i][j] = 0 if $(i, j) \notin E$

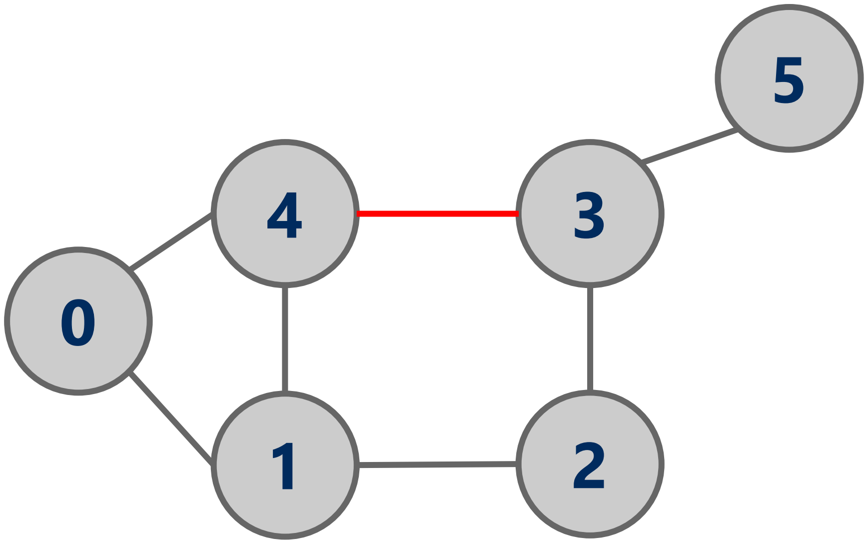
	0	1	2	3	4	5
0	0	1	0	0	1	0
1	1	0	1	0	1	0
2	0	1	0	1	0	0
3	0	0	1	0	1	1
4	1	1	0	1	0	0
5	0	0	0	1	0	0



Using an adjacency matrix

- Rows/columns are vertex labels
  - M[i][j] = 1 if (i, j) ∈ E
  - M[i][j] = 0 if (i, j) ∉ E

	0	1	2	3	4	5
0	0	1	0	0	1	0
1	1	0	1	0	1	0
2	0	1	0	1	0	0
3	0	0	1	0	1	1
4	1	1	0	1	0	0
5	0	0	0	1	0	0



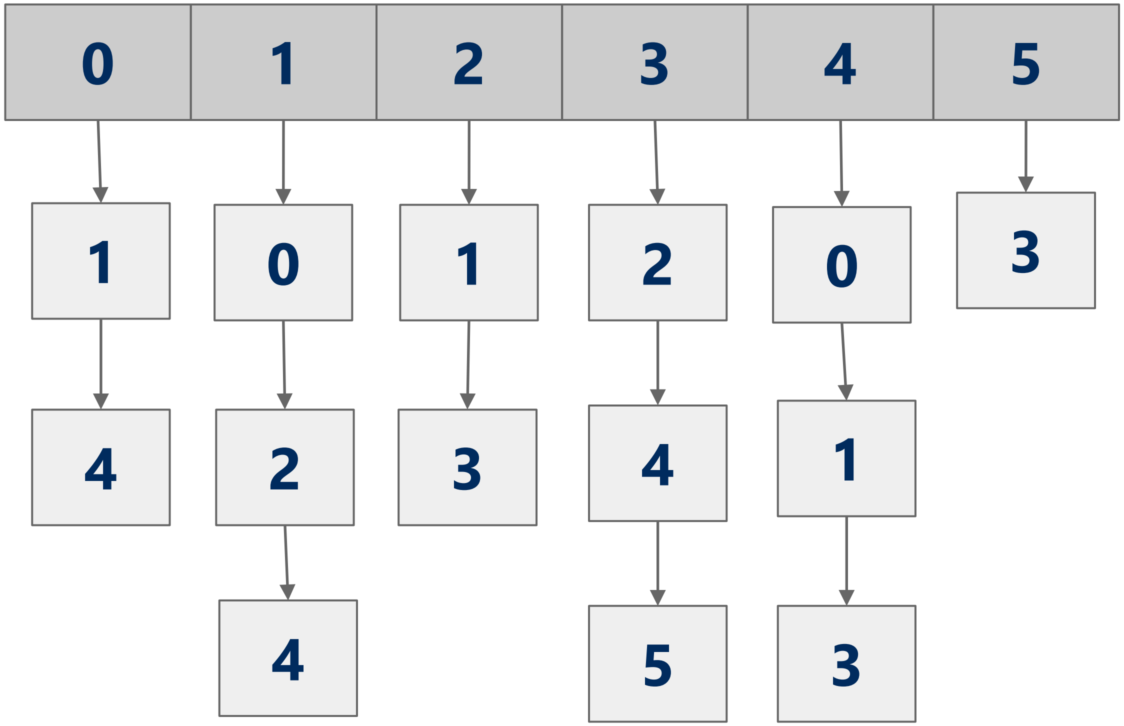
Adjacency matrix analysis

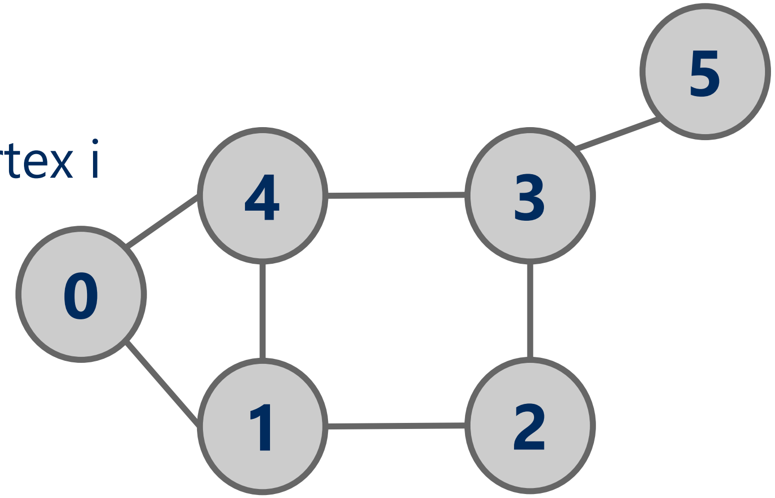
- Runtime?
  - Check if two vertices are neighbors
    - $\Theta(1)$
  - Find the list of neighbors of a vertex
    - $O(v)$
  - $O(v^2)$ time to initialize
- Space?
  - $O(v^2)$

	0	1	2	3	4	5
0	0	1	0	0	1	0
1	1	0	1	0	1	0
2	0	1	0	1	0	0
3	0	0	1	0	1	1
4	1	1	0	1	0	0
5	0	0	0	1	0	0

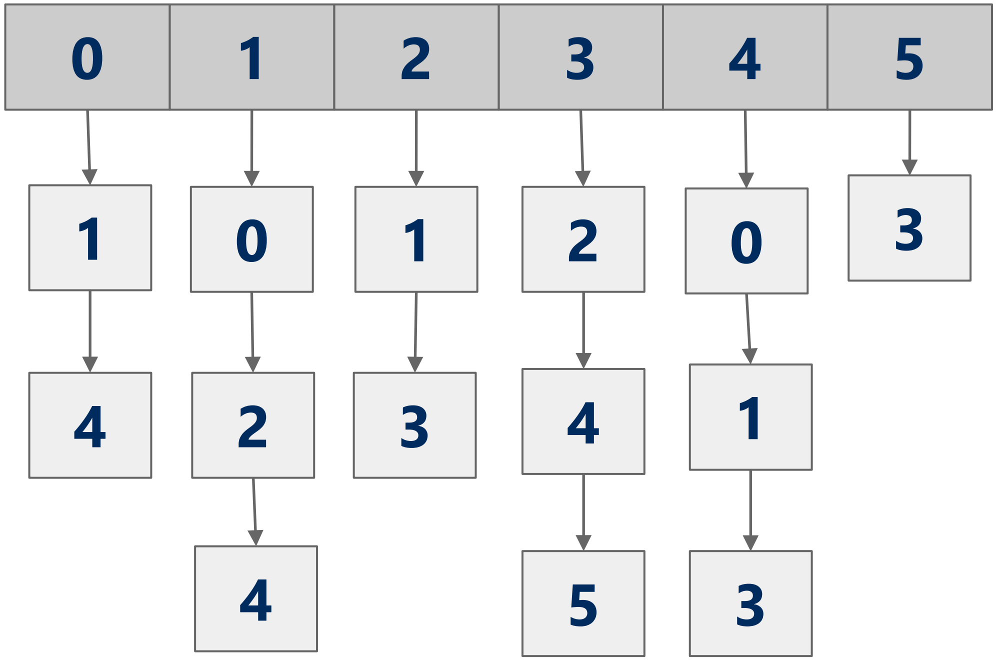
Adjacency lists

- Array of neighbor lists
  - A[i] contains a list of the neighbors of vertex i





Adjacency list analysis



- Runtime?
  - Check if two vertices are neighbors
  - Find the list of neighbors of a vertex
    - Θ(d)
    - d is the degree of a vertex (# of neighbors)
    - O(v)
- Space?
  - Θ(v + e) memory
  - overhead of node use
  - Could be much less than v^2

Comparison

- **Where would we want to use adjacency lists vs adjacency matrices?**
- Dense graphs?
- Sparse graphs?
- **What about the list of vertices/list of edges approach?**

Even more definitions

- Path
  - A sequence of adjacent vertices
- Simple Path
  - A path in which no vertices are repeated
- Simple Cycle
  - A simple path with the same first and last vertex
- Connected Graph
  - A graph in which a path exists between all vertex pairs
- Connected Component
  - Connected subgraph of a graph
- Acyclic Graph
  - A graph with no cycles
- Tree
  - ?
  - A connected, acyclic graph
    - Has exactly v-1 edges

Complete Graph vs. Connected Graph

- Difference between Connected graph and Complete graph?
  - Connected means there is a **path** from A to B for each pair of vertices A and B
  - Complete means there is an **edge** between A and B for each pair of vertices A and B

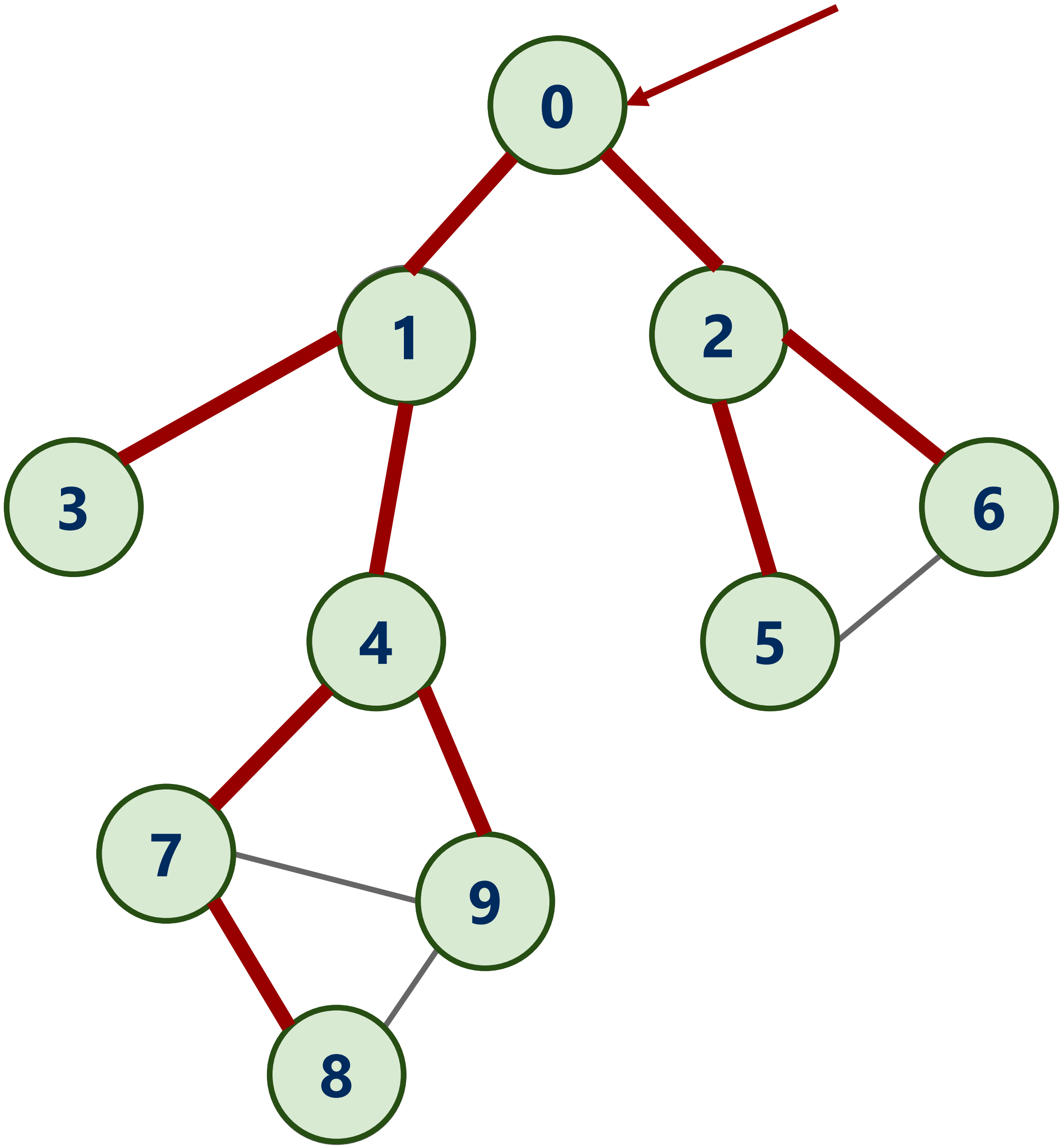
Graph traversal

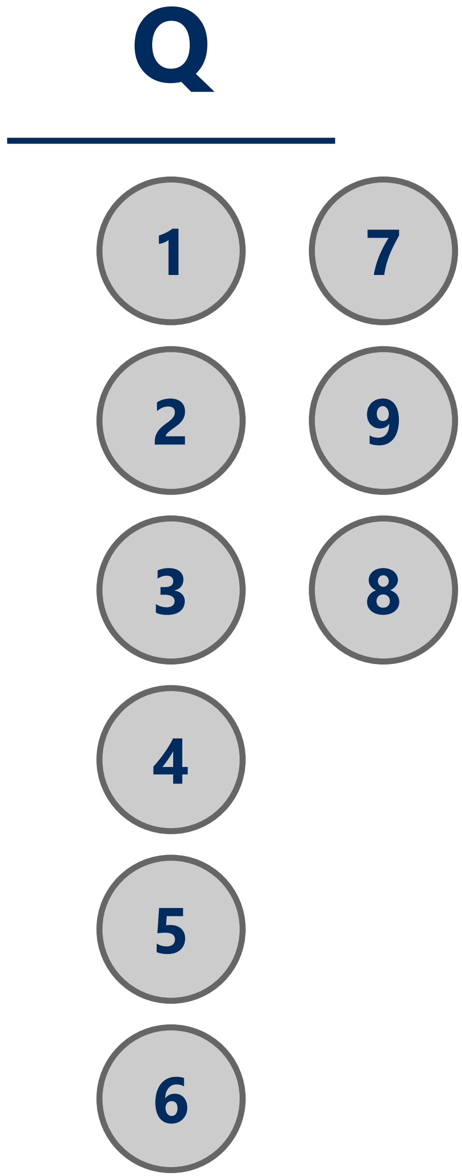
- What is the best order to traverse a graph?
- Two primary approaches:
  - Breadth-first search (BFS)
    - Search all directions evenly
      - i.e., from i, visit all of i's neighbors, then all of their neighbors, etc.
    - Would help us compute the distance between two vertices
      - Remember our Problem of the Day?
  - Depth-first search (DFS)
    - "Dive" as deep as possible into the graph first
    - Branch when necessary

BFS

- Can be easily implemented using a queue
  - For each vertex visited, add all of its neighbors to the Q (if not previously added)
    - Vertices that have been seen (i.e., added to the Q) but not yet visited are said to be the *fringe*
  - Pop head of the queue to be the next visited vertex
- See example

BFS example





BFS Pseudo-code

```
Q = new Queue

BFS(vertex v){
    add v to Q
    while(Q is not empty){
        w = remove head of Q
        visited[w] = true //mark w as visited
        for each unseen neighbor x
            seen[x] = true //mark x as seen
            parent[x] = w
            add x to Q
    }
}
```

Shortest paths

- BFS traversals can further be used to determine the *shortest path* between two vertices

BFS Pseudo-code to compute shortest paths

```
Q = new Queue

BFS(vertex v){
    add v to Q
    while(Q is not empty){
        w = remove head of Q
        visited[w] = true //mark w as visited
        for each unseen neighbor x
            seen[x] = true //mark x as seen
            parent[x] = w
            distance[x] = distance[w] + 1
            add x to Q
    }
}
```

Problem of the Day

- **Input**: A file containing LinkedIn Connection information formatted like the following:
  - Account1: Connection1, Connection2, …
  - Account2: Connection1, Connection2, …
  - …
- **Output**: Answer the following questions:
  - Given two LI accounts, how "far" are they from each other?
    - E.g., 1st connection, 2nd connection, etc.
  - Are the accounts in the file all ***connected***?
    - If not, how many ***connected components*** are there?
  - Are there certain accounts that if removed, the remaining accounts become ***partitioned***?
    - These account are called ***articulation points***

Problem of the Day

- **Input**: A file containing LinkedIn Connection information formatted like the following:
  - Account1: Connection1, Connection2, …
  - Account2: Connection1, Connection2, …
  - …
- **Output**: Answer the following questions:
  - Given two LI accounts, how "far" are they from each other?
    - E.g., 1st connection, 2nd connection, etc.
  - Are the accounts in the file all ***connected***?
    - If not, how many ***connected components*** are there?
  - Are there certain accounts that if removed, the remaining accounts become ***partitioned***?
    - These account are called ***articulation points***

Finding connected components

- A connected component is a connected subgraph G′
  - (V′, E′)
    - V′ ⊆ V
    - E′ = {(u, v) ∈ E and both u and v ∈ V′}
- To find all connected components:
  - wrapper function around BFS
  - A loop in the wrapper function will have to continually call bfs() while **there are still unseen vertices**
  - Each call will yield a spanning tree for a **connected component** of the graph

BFS Pseudo-code to compute connected components

```
int components = 0
for each vertex v in V
        if visited[v] = false
                components++
                Q = new Queue
                BFS(v)
```

```
BFS(vertex v){
        add v to Q
        component[v] = components
        component
        while(Q is not empty){
                w = remove head of Q
                visited[w] = true
                for each unseen neighbor x
                        seen[x] = true
                        add x to Q
        }
}
```

Problem of the Day

- **Input**: A file containing LinkedIn Connection information formatted like the following:
  - Account1: Connection1, Connection2, …
  - Account2: Connection1, Connection2, …
  - …
- **Output**: Answer the following questions:
  - Given two LI accounts, how "far" are they from each other?
    - E.g., 1^{st} connection, 2^{nd} connection, etc.
  - Are the accounts in the file all ***connected***?
    - If not, how many ***connected components*** are there?
  - Are there certain accounts that if removed, the remaining accounts become ***partitioned***?
    - These account are called ***articulation points***

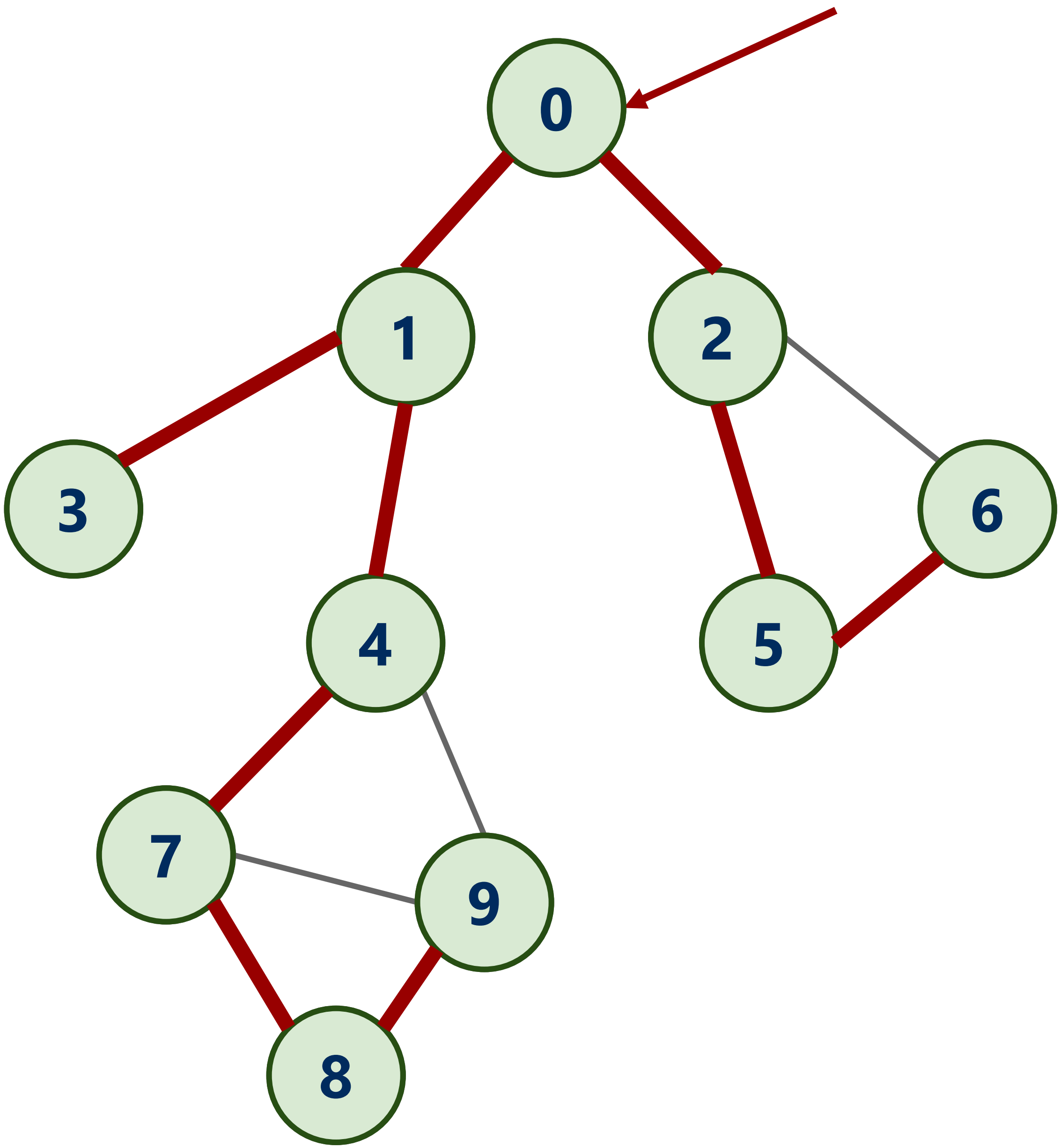
DFS – Depth First Search

- Already seen and used this throughout the term
  - For Huffman encoding…
- Can be easily implemented recursively
  - For each vertex, visit first (in some arbitrary order) unseen neighbor
    - recursively!
  - Backtrack at deadends (i.e., vertices with no unseen neighbors)
    - Try next unseen neighbor after backtracking

DFS Pseudo-code

```
DFS(vertex v) {
  seen[v] = true //mark v as seen
  for each unseen neighbor w
    parent[w] = v
    DFS(w)
}
```

DFS example 2



When to visit a vertex

```
DFS(vertex v) {
  seen[v] = true //mark v as seen
  visit v //before visiting children in the spanning tree
  for each unseen neighbor w
    parent[w] = v
    DFS(w)
}
```

When to visit a vertex

```
DFS(vertex v) {
  seen[v] = true //mark v as seen
for each unseen neighbor w
    parent[w] = v
    DFS(w)
 visit v //after visiting children in the spanning tree
}
```

When to visit a vertex

```
DFS(vertex v) {
   seen[v] = true //mark v as seen
for each unseen neighbor w
      parent[w] = v
      visit v //between visiting children in the spanning tree
      DFS(w)
}
```